1. What is the name of the feature responsible for generating Regex objects?

Ans - The name of the feature responsible for generating Regex objects in Python is the re.compile() function.

This function takes a regular expression pattern as an argument and returns a Regex object that can be used for various operations, such as searching, matching, replacing, or splitting strings

1. Why do raw strings often appear in Regex objects?

Ans - Raw strings are often used in Regex objects in Python because they prevent the Python interpreter from interpreting the backslash \ as an escape character. Escape characters are special characters that have a different meaning when preceded by a backslash, such as \n for newline or \t for tab.

However, in regular expressions, the backslash is also used to indicate special sequences, such as \d for digits or \w for words1.

Therefore, if we use a normal string to create a Regex object, we have to escape the backslash twice: once for Python and once for the regular expression engine.

1. What is the return value of the search() method?

Ans - The return value of the search() method in Python is a Match object.

A Match object contains information about the successful match of a regular expression pattern in a string.

If the search() method does not find a match, it returns None instead of a Match object

1. From a Match item, how do you get the actual strings that match the pattern?

Ans- To get the actual strings that match the pattern from a Match object in Python, we can use the group() method.

The group() method returns the matched substring as a string.

We can also pass an argument to the group() method to specify which group we want to return,

if the pattern contains capturing parentheses.

For example:

# Import the re module

import re

# Define a pattern and a string

pattern = r'(\w+)@(\w+)\.(\w+)'

string = 'Email: john@example.com'

# Search for the pattern in the string

match = re.search(pattern, string)

# Check if a match was found

if match:

# Print the entire matched string

print(match.group()) # 'john@example.com'

# Print the first group (the user name)

print(match.group(1)) # 'john'

# Print the second group (the domain name)

print(match.group(2)) # 'example'

# Print the third group (the top-level domain)

print(match.group(3)) # 'com'

else:

# No match was found

print('No match found')

1. In the regex which created from the r'(\d\d\d)-(\d\d\d-\d\d\d\d)', what does group zero cover? Group 2? Group 1?

Ans - A regular expression (regex) is a pattern of text that can be used to search for or manipulate strings.

In Python, the re module provides functions and methods for working with regexes.

A regex can contain groups, which are parts of the pattern enclosed in parentheses.

Groups can be used to extract or modify specific parts of the matched string.

In the regex r'(\d\d\d)-(\d\d\d-\d\d\d\d)', there are two groups: (\d\d\d) and (\d\d\d-\d\d\d\d).

The \d is a special sequence that matches any digit character.

The r before the string indicates that it is a raw string, which means that backslashes are not treated as escape characters by Python.

When a regex matches a string, it creates a Match object that contains information about the match.

The Match object has a group() method that can return the matched substring or a specific group from the match.

The groups are numbered from left to right, starting from 1. Group 0 is the entire match.

For example, if the regex r'(\d\d\d)-(\d\d\d-\d\d\d\d)' matches the string '415-555-4242', then:

Group 0 covers the entire match, '415-555-4242'.

Group 1 covers the first set of parentheses, '415'.

Group 2 covers the second set of parentheses, '555-4242'.

We can use the group() method with an argument to return a specific group.

For example:

>>> import re

>>> pattern = r'(\d\d\d)-(\d\d\d-\d\d\d\d)'

>>> string = '415-555-4242'

>>> match = re.search(pattern, string)

>>> match.group(0)

'415-555-4242'

>>> match.group(1)

'415'

>>> match.group(2)

'555-4242'

1. In standard expression syntax, parentheses and intervals have distinct meanings. How can you tell a regex that you want it to fit real parentheses and periods?

Ans - To tell a regex that we want it to match real parentheses and periods in Python, we have to escape them with a backslash \.

Escaping means that we tell the regex engine to treat the following character as a literal character, not as a special character with a different meaning.

Parentheses () and periods . are special characters in regexes, because they have different functions.

Parentheses are used to create groups, which are parts of the pattern that can be extracted or modified separately.

Periods are used to match any single character, except for newline characters.

For example, if we want to match the string 'Hello (world).', we have to escape the parentheses and the period with backslashes, like this: r'Hello\\ (world\\)\\.'. The r before the string indicates that it is a raw string, which means that Python will not interpret the backslashes as escape characters.

If we don’t escape the parentheses and the period, the regex will match any string that starts with 'Hello', followed by any single character, followed by 'world', followed by any single character.

For example-

# Import the re module

import re

# Define a pattern and a string

pattern = r'Hello\ (world)\.'

string = 'Hello (world).'

# Search for the pattern in the string

match = re.search(pattern, string)

# Check if a match was found

if match:

# Print the matched string

print(match.group()) # 'Hello (world).'

# Print the first group (the word inside the parentheses)

print(match.group(1)) # 'world'

else:

# No match was found

print('No match found')

==============================================================================

1. The findall() method returns a string list or a list of string tuples. What causes it to return one of the two options?

Ans – The findall() method in Python returns a list of strings or a list of tuples depending on the number of groups in the regular expression pattern.

A group is a part of the pattern that is enclosed in parentheses.

Groups can be used to extract or modify specific parts of the matched string.

If the pattern has no groups, the findall() method returns a list of strings that match the entire pattern.

For example, if the pattern is r'\d+' (one or more digits), and the string is 'abc123def456', the findall() method returns ['123', '456'].

If the pattern has one group, the findall() method returns a list of strings that match the group.

For example, if the pattern is r'\d+ (\w+)' (one or more digits followed by a space and one or more word characters), and the string is '123 abc 456 def', the findall() method returns ['abc', 'def'].

If the pattern has more than one group, the findall() method returns a list of tuples that contain the strings that match each group.

For example, if the pattern is r'(\d+) (\w+)' (same as above, but with both parts in groups), and the string is '123 abc 456 def', the findall() method returns [('123', 'abc'), ('456', 'def')].

==============================================================================

1. In standard expressions, what does the | character mean?

Ans – In standard expressions, also known as regular expressions or regexes, the | character means alternation.

It is used to match one of several possible subexpressions in a pattern.

For example, the pattern r'cat|dog' matches either 'cat' or 'dog' in a string.

The | character can also be used inside parentheses to create groups of alternatives.

For example, the pattern r'(red|green|blue) car' matches either 'red car', 'green car', or 'blue car' in a string.

==============================================================================

1. In regular expressions, what does the character stand for?

Ans - The character is a metacharacter in regular expressions, which means that it has a special meaning and does not match itself.

The character is used to indicate the end of a string or a line.

For example, the pattern r'world$' matches the word 'world' only if it is at the end of the string or the line.

The character can also be used in conjunction with other metacharacters, such as ^ (start of string or line) or | (alternation), to create more complex patterns.

==============================================================================

10.In regular expressions, what is the difference between the + and \* characters?

Ans- The + and \* characters are quantifiers in regular expressions, which means that they specify how many times the preceding element can be repeated in a match. The + character means one or more, while the \* character means zero or more.

For example:

The pattern r'a+' matches one or more occurrences of the letter ‘a’ in a string. It can match 'a', 'aa', 'aaa', etc., but not an empty string.

The pattern r'a\*' matches zero or more occurrences of the letter ‘a’ in a string. It can match 'a', 'aa', 'aaa', etc., as well as an empty string.

The pattern r'[0-9]+' matches one or more digits in a string. It can match '1', '12', '123', etc., but not an empty string or a non-digit character.

The pattern r'[0-9]\*' matches zero or more digits in a string. It can match '1', '12', '123', etc., as well as an empty string or a non-digit character.

The difference between the + and \* characters is that the + character requires at least one occurrence of the preceding element, while the \* character allows zero occurrences. This can affect the result of the match, especially when using methods like re.search() or re.findall().

==============================================================================

1. What is the difference between {4} and {4,5} in regular expression?

Ans - {4} and {4,5} are quantifiers in regular expression, which means that they specify how many times the preceding element can be repeated in a match.

{4} means exactly four, while {4,5} means between four and five, inclusive.

For example, the pattern r'\d{4}' matches a string that contains exactly four digits, such as '1234' or '5678', but not '123' or '12345'.

The pattern r'\d{4,5}' matches a string that contains four or five digits, such as '1234', '5678', '12345', or '67890', but not '123' or '123456'.

We can use these quantifiers to create more complex patterns that match specific formats or ranges of values.

For example, the pattern r'\d{4}-\d{4}-\d{4}-\d{4}' matches a credit card number that consists of four groups of four digits separated by hyphens,

such as '1111-2222-3333-4444'.

The pattern r'[A-Z]{2}\d{2}[A-Z]{2}' matches a vehicle registration number that consists of two letters, two digits, and two letters, such as 'AB12CD'

==============================================================================

1. What do you mean by the \d, \w, and \s shorthand character classes signify in regular expressions?

ANS – Regular expressions, or regexes, are patterns of text that can be used to search for or manipulate strings.

Python has a built-in module called re that provides functions and methods for working with regexes.

Shorthand character classes are a way of simplifying regexes by using a single character to represent a set of characters.

For example, instead of writing [0-9] to match any digit, you can write \d.

The \d, \w, and \s shorthand character classes signify the following sets of characters:

\d matches any digit character, which means 0-9.

\w matches any word character, which means any lowercase letter, any uppercase letter, the underscore \_, or any digit.

\s matches any whitespace character, which includes the space , the tab \t, the carriage return \r, the line feed \n, or the form feed \f.

We can use these shorthand character classes both inside and outside square brackets.

For example, \d+ matches one or more digits, while [\d\s] matches a single character that is either a digit or a whitespace.

==============================================================================

1. What do means by \D, \W, and \S shorthand character classes signify in regular expressions?

ANS – Regular expressions, or regexes, are patterns of text that can be used to search for or manipulate strings.

Python has a built-in module called re that provides functions and methods for working with regexes.

Shorthand character classes are a way of simplifying regexes by using a single character to represent a set of characters.

For example, instead of writing [0-9] to match any digit, you can write \d.

The \D, \W, and \S shorthand character classes signify the following sets of characters:

\D matches any character that is not a digit (0-9).

\W matches any character that is not a word character. A word character is defined as any letter, digit, or underscore (\_).

\S matches any character that is not a whitespace character. A whitespace character includes the space , the tab \t, the carriage return \r, the line feed \n, or the form feed \f.

We can use these shorthand character classes both inside and outside square brackets.

For example, \D+ matches one or more non-digits, while [\D\S] matches a single character that is either not a digit or not a whitespace.

==============================================================================

1. What is the difference between .\*? and .\*?

ANS - .? and .? are quantifiers in regular expressions, which means that they specify how many times the preceding element can be repeated in a match.

The . character matches any single character, except for newline characters.

.\*? means zero or more, but as few as possible. It is also called a lazy or non-greedy quantifier, because it tries to match the smallest possible number of characters that satisfy the pattern.

.\* means zero or more, but as many as possible. It is also called a greedy quantifier, because it tries to match the largest possible number of characters that satisfy the pattern.

The difference between .? and . can affect the result of the match, especially when using methods like re.search() or re.findall().

For example:

# Import the re module

import re

# Define a pattern and a string

pattern = r'<.\*?>'

string = '<p>Hello</p><p>World</p>'

# Search for the pattern in the string

match = re.search(pattern, string)

# Check if a match was found

if match:

# Print the matched string

print(match.group()) # '<p>'

else:

# No match was found

print('No match found')

In this example, the pattern r'<.\*?>' matches any substring that starts with < and ends with >, but as few characters as possible.

Therefore, it matches only the first <p> tag in the string.

# Import the re module

import re

# Define a pattern and a string

pattern = r'<.\*>'

string = '<p>Hello</p><p>World</p>'

# Search for the pattern in the string

match = re.search(pattern, string)

# Check if a match was found

if match:

# Print the matched string

print(match.group()) # '<p>Hello</p><p>World</p>'

else:

# No match was found

print('No match found')

In this example, the pattern r'<.\*>' matches any substring that starts with < and ends with >, but as many characters as possible.

Therefore, it matches the entire string, from the first < to the last >.

==============================================================================

1. What is the syntax for matching both numbers and lowercase letters with a character class?

ANS- A character class is a way of specifying a set of characters that we want to match in a regular expression.

We can use square brackets [ ] to create a character class.

To match both numbers and lowercase letters, we can use the character class [a-z0-9].

This means any lowercase letter from a to z or any digit from 0 to 9.

We can also use ranges of letters or numbers by using a hyphen.

For example, the character class [a-zA-Z0–9] will match all lowercase letters, uppercase letters, and numbers.

We can use quantifiers to specify how many times the character class can be repeated in a match.

For example, [a-z0-9]+ means one or more occurrences of the character class, while [a-z0-9]\* means zero or more occurrences.

==============================================================================

1. What is the procedure for making a normal expression in regax case insensitive?

ANS- A normal expression, or a regular expression, or a regex, is a pattern of text that can be used to search for or manipulate strings.

Python has a built-in module called re that provides functions and methods for working with regexes.

To make a normal expression case insensitive, we can pass re.I or re.IGNORECASE as the second argument to re.compile().

This will create a regex object that ignores the case of the letters when matching.

For example:

# Import the re module

import re

# Define a normal expression and a string

pattern = 'hello'

string = 'Hello World'

# Compile the normal expression with re.I flag

regex = re.compile(pattern, re.I)

# Search for the normal expression in the string

match = regex.search(string)

# Check if a match was found

if match:

# Print the matched string

print(match.group()) # 'Hello'

else:

# No match was found

print('No match found')

We can also use inline modifiers to make part of the normal expression case insensitive.

The syntax is (?i:...), where ... is the part of the pattern that you want to make case insensitive.

For example:

# Import the re module

import re

# Define a normal expression and a string

pattern = r'(?i:hello) world'

string = 'Hello World'

# Search for the normal expression in the string

match = re.search(pattern, string)

# Check if a match was found

if match:

# Print the matched string

print(match.group()) # 'Hello World'

else:

# No match was found

print('No match found')

==============================================================================

1. What does the . character normally match? What does it match if re.DOTALL is passed as 2nd argument in re.compile()?

ANS- The . character is a metacharacter in regexes, which means that it has a special meaning and does not match itself.

The . character matches any single character, except for newline characters.

If re.DOTALL is passed as the second argument to re.compile(), then the . character will also match newline characters.

This means that the . character will match any character, including newline characters.

For example, if the pattern is r'.\*' (zero or more occurrences of any character), and the string is 'Hello\nWorld', then:

Without re.DOTALL, the . character will not match the newline character \n, and the pattern will match only 'Hello'.

With re.DOTALL, the . character will match the newline character \n, and the pattern will match the entire string 'Hello\nWorld'.

==============================================================================

1. If numReg = re.compile(r'\d+'), what will numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen') return?

ANS – The re.compile() function takes a regex pattern as an argument and returns a regex object that can be used for various operations, such as searching, matching, replacing, or splitting strings.

The \d is a special sequence in regexes that matches any digit character, which means 0-9.

The + is a quantifier in regexes that means one or more occurrences of the preceding element.

The sub() method of a regex object takes two arguments: a replacement string and a target string.

It returns a new string where all the matches of the regex pattern in the target string are replaced by the replacement string.

Therefore, if numReg = re.compile(r'\d+'), then the regex pattern is one or more digits.

If numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen') is executed, then it will return a new string where all the substrings that consist of one or more digits in the target string are replaced by 'X'.

The result will be 'X drummers, X pipers, five rings, X hen'.

==============================================================================

1. What does passing re.VERBOSE as the 2nd argument to re.compile() allow to do?

ANS – The re.VERBOSE argument allows you to add whitespace and comments to the string passed to re.compile().

This makes the regex pattern more readable and understandable by allowing you to visually separate logical sections of the pattern and add explanations.

Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like \*?, (?: or (?P<…>). When a line contains a # that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such # through the end of the line are ignored.

For example, you can write a regex pattern to match an email address like this:

# Import the re module

import re

# Define a regex pattern with re.VERBOSE

pattern = re.compile(r"""

^ # Start of string

( # Capture group for the local part

[a-z0-9\_.-]+ # One or more alphanumeric characters, underscore, dot, or hyphen

)

@ # Literal @ symbol

( # Capture group for the domain name

[a-z0-9.-]+ # One or more alphanumeric characters, dot, or hyphen

)

\. # Literal dot symbol

( # Capture group for the top-level domain

[a-z]{2,6} # Two to six alphabetic characters

)

$ # End of string

""", re.VERBOSE | re.IGNORECASE) # Combine re.VERBOSE with re.IGNORECASE to ignore case

# Define a string to test the pattern

string = "john@example.com"

# Search for the pattern in the string

match = pattern.search(string)

# Check if a match was found

if match:

# Print the matched string and the groups

print("Matched string:", match.group())

print("Local part:", match.group(1))

print("Domain name:", match.group(2))

print("Top-level domain:", match.group(3))

else:

# No match was found

print("No match found")

Output:

Matched string: john@example.com

Local part: john

Domain name: example

Top-level domain: com

==============================================================================

20. How would you write a regex that match a number with comma for every three digits? It must match the given following:

'42'

'1,234'

'6,368,745'

but not the following:

'12,34,567' (which has only two digits between the commas)

'1234' (which lacks commas)

ANS – To write a regex that match a number with comma for every three digits, we can use the following pattern:

pattern = r'^\d{1,3}(,\d{3})\*$'

This pattern means the following:

^ matches the start of the string.

\d{1,3} matches one to three digits.

(,\d{3})\* matches zero or more occurrences of a comma followed by three digits.

$ matches the end of the string.

This pattern will match strings that consist of one to three digits, optionally followed by one or more groups of a comma and three digits.

It will not match strings that have more than three digits before the first comma, or less than three digits after any comma.

For example, this pattern will match '42', '1,234', and '6,368,745', but not '12,34,567' or '1234'.

We can use the re module to create a regex object from this pattern and use its methods to perform various operations on strings.

For example:

# Import the re module

import re

# Create a regex object from the pattern

numRegex = re.compile(r'^\d{1,3}(,\d{3})\*$')

# Define some strings to test the pattern

strings = ['42', '1,234', '6,368,745', '12,34,567', '1234']

# Loop through the strings and check if they match the pattern

for string in strings:

# Use the search() method to find a match in the string

match = numRegex.search(string)

# If a match is found, print the matched string

if match:

print(match.group())

# If no match is found, print 'No match'

else:

print('No match')

Output:

42

1,234

6,368,745

No match

No match

==============================================================================

21. How would you write a regex that matches the full name of someone whose last name is Watanabe? You can assume that the first name that comes before it will always be one word that begins with a capital letter. The regex must match the following:

'Haruto Watanabe'

'Alice Watanabe'

'RoboCop Watanabe'

but not the following:

'haruto Watanabe' (where the first name is not capitalized)

'Mr. Watanabe' (where the preceding word has a nonletter character)

'Watanabe' (which has no first name)

'Haruto watanabe' (where Watanabe is not capitalized)

ANS-

To write a regex that matches the full name of someone whose last name is Watanabe and the first name is a capitalized word,

we can use the following pattern:

pattern = r'^[A-Z][a-z]\*\sWatanabe$'

This pattern means the following:

^ matches the start of the string.

[A-Z] matches any uppercase letter from A to Z.

[a-z]\* matches zero or more lowercase letters from a to z.

\s matches any whitespace character, such as a space or a tab.

Watanabe matches the literal string ‘Watanabe’.

$ matches the end of the string.

This pattern will match strings that consist of a capitalized word followed by a whitespace and then ‘Watanabe’.

It will not match strings that have no first name, or have a non-capitalized first or last name, or have a non-letter character before the last name.

For example, this pattern will match ‘Haruto Watanabe’, ‘Alice Watanabe’, and ‘RoboCop Watanabe’, but not ‘haruto Watanabe’, ‘Mr. Watanabe’, ‘Watanabe’, or ‘Haruto watanabe’.

We can use the re module to create a regex object from this pattern and use its methods to perform various operations on strings.

For example:

# Import the re module

import re

# Create a regex object from the pattern

nameRegex = re.compile(r'^[A-Z][a-z]\*\sWatanabe$')

# Define some strings to test the pattern

strings = ['Haruto Watanabe', 'Alice Watanabe', 'RoboCop Watanabe', 'haruto Watanabe', 'Mr. Watanabe', 'Watanabe', 'Haruto watanabe']

# Loop through the strings and check if they match the pattern

for string in strings:

# Use the search() method to find a match in the string

match = nameRegex.search(string)

# If a match is found, print the matched string

if match:

print(match.group())

# If no match is found, print 'No match'

else:

print('No match')

Output:

Haruto Watanabe

Alice Watanabe

RoboCop Watanabe

No match

No match

No match

No match

==============================================================================

22. How would you write a regex that matches a sentence where the first word is either Alice, Bob, or Carol; the second word is either eats, pets, or throws; the third word is apples, cats, or baseballs; and the sentence ends with a period? This regex should be case-insensitive. It must match the following:

'Alice eats apples.'

'Bob pets cats.'

'Carol throws baseballs.'

'Alice throws Apples.'

'BOB EATS CATS.'

but not the following:

'RoboCop eats apples.'

'ALICE THROWS FOOTBALLS.'

'Carol eats 7 cats.'

ANS –

We can use the following pattern:

pattern = r'^(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.$'

This pattern means the following:

^ matches the start of the string.

(Alice|Bob|Carol) matches either ‘Alice’, ‘Bob’, or ‘Carol’. The parentheses create a group of alternatives, and the vertical bar | means ‘or’.

\s matches any whitespace character, such as a space or a tab.

(eats|pets|throws) matches either ‘eats’, ‘pets’, or ‘throws’. This is another group of alternatives.

(apples|cats|baseballs) matches either ‘apples’, ‘cats’, or ‘baseballs’. This is the third group of alternatives.

\. matches a literal dot character. The backslash \ escapes the dot, which is a special character in regexes that means any character except newline.

$ matches the end of the string.

To make this pattern case insensitive, we can pass re.I or re.IGNORECASE as the second argument to re.compile().

This will create a regex object that ignores the case of the letters when matching.

For example:

# Import the re module

import re

# Create a regex object from the pattern with re.I flag

regex = re.compile(r'^(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.$', re.I)

# Define some strings to test the pattern

strings = ['Alice eats apples.', 'Bob pets cats.', 'Carol throws baseballs.', 'Alice throws Apples.', 'BOB EATS CATS.', 'RoboCop eats apples.', 'ALICE THROWS FOOTBALLS.', 'Carol eats 7 cats.']

# Loop through the strings and check if they match the pattern

for string in strings:

# Use the search() method to find a match in the string

match = regex.search(string)

# If a match is found, print the matched string

if match:

print(match.group())

# If no match is found, print 'No match'

else:

print('No match')

Output:

Alice eats apples.

Bob pets cats.

Carol throws baseballs.

Alice throws Apples.

BOB EATS CATS.

No match

No match

No match

==============================================================================